Behavior-driven Integration Testing S. 24
Angular-2-Security-Architektur in der Praxis S. 64
REST-Services auf Basis von Spring Boot S. 84

JAVA Mag
Javamagazin™
Java | Architektur | Software-Innovation

KOTLIN

The new Kid on the JVM

Ausgabe 3.2017

Deutschland € 9,80
Österreich € 10,80
Schweiz sFr 19,50
Luxemburg € 11,15

4 194586 709801 03

www.javamagazin.de

Teil 3: Optimierungsmaßnahmen gezielt umsetzen

Änderungen in die Produktion bringen

Im letzten Teil dieser Miniserie haben wir aufgezeigt, welche Vorbereitungen wir treffen können, um Änderungen am produktiv eingespielten Softwarestand erfolgreich durchführen zu können. Nun geht es an die Optimierungsmaßnahmen – denn es geht immer noch ein bisschen besser.

von Joachim Arrasz und Sebastian Heib

In den letzten Teilen dieser Serie waren wir als Datensammler und -analysten unterwegs. Diese Daten haben wir über verschiedene Tools verdichtet, analysiert und visualisiert. Wir haben das gemacht, damit wir einen Trend haben, gegen den wir unsere Erweiterungen, Verbesserungen oder auch Fehlerbehebungen prüfen können. Weiterhin erhält man so eine gute Vorhersagemöglichkeit für die Stabilität, die der Grund für den Aufwand ist. In diesem Teil werden wir aufzeigen, mit welchen Methoden man Bugfixes, Changes, weitere Features, aber auch absolut dringende Hotfixes am besten in das System integriert. Für das bessere Verständnis wollen wir das anhand einer kleinen fiktiven Anwendung darstellen. Diese Anwendung verwaltet den Fuhrpark der SYNGO-Autovermietung (**Abb. 1**). Die Anwendungsfälle sind sehr einfach: Anzeige des kompletten Fuhrparks, Anzeige der Verfügbarkeit eines konkreten Fahrzeugs sowie Buchen eines Fahrzeugs für einen bestimmten Zeitraum.

Gehen wir nun davon aus, dass das Entwicklungsteam die Version 1.0 der Software nach den Vorgaben dieser Serie zur Produktion bereitgestellt hat, wobei Git als Versionsverwaltung verwendet wird und die Versionsnummer zum Release fix getaggt wurde. Dann wäre es nun an der Zeit, sich zu fragen, was für den Kunden die höchste Priorität hat. Das Wichtigste ist, die Stabilität der Software zu gewährleisten. Entscheidende Grundlagen hierfür sind die schon erwähnte Versionsverwaltung mit der sauberen Versionierung der verschiedenen Softwarestände sowie dem Vorhandensein von Tags für fertige Versionen. Unterstützung hierfür bietet z. B. das Maven-Release-Plug-in, um Releases immer nach dem gleichen Schema durchzuführen. In einem weiteren Schritt kann weiter automatisiert werden, indem diese Releases beispielsweise durch die CI-Umgebung getriggert werden. Das Wichtigste ist, die Stabilität der Software zu gewährleisten, in Abhängigkeit der notwendigen Dinge, die eigentlich zu tun sind: Grundlagen wie einheitliche Versionsverwaltung, saubere Versionierung und das Setzen von Tags durch Automatisierung der Deployments, beispielsweise über das Maven-Release-Plug-in.

Ein guter Ansatz zur Versionsverwaltung ist dabei die klassische Unterteilung in Major-, Minor- und Bugfix-

Artikelserie

Teil 1: Stabile Produktion und Betrieb von Software in der Industrie
Teil 2: Nachhaltige Analyse gesammelter Daten zur Projektunterstützung
Teil 3: Optimierungsmaßnahmen gezielt umsetzen

Deployment-Zeitpunkt

Für die Stabilität eines Betriebs ist es unumgänglich, sich auch Gedanken um den Zeitpunkt eines Deployments zu machen. Es gibt verschiedene Faktoren, die man hier berücksichtigen muss.
Zum einen die Branche, in der man sich befindet. Nehmen wir die Langzeitplanung. Viele Branchen haben Spitzenzeiten, in denen ihr Geschäft reibungslos laufen muss, z. B. die Spielzeugindustrie zur Weihnachtszeit. Hier sollte man zu dieser Zeit eher kein Deployment einplanen.
Zum anderen Trends, die man aus den Statistiken ableiten konnte: Für die kurzzeitige Planung während eines Monats, einer Woche und dann auch des Tages kann man sehr gut die Trends aus dem Monitoring heranziehen. Daraus kann man ableiten, wann ein Deployment am wenigsten stört.

Release-Nummern (z. B. 1.4.2 – Major 1, Minor 4, Bugfix 2). Dabei wird die Bugfix-Nummer immer dann hochgezählt, wenn eine neue Version der Software ausgeliefert wird, die nur Bugs fixt – gegebenenfalls sogar nur einen einzelnen. Die Minor-Nummer wird verändert, sobald neue Features oder Changes an bestehenden Features ausgerollt werden. Und die Major-Nummer wird letztlich nur dann erhöht, wenn sich die Kompatibilität von Schnittstellen ändert – also dann, wenn fremde Systeme ihre Schnittstellen entsprechend anpassen müssten. Solange die Major-Nummer gleich bleibt, kann man also davon ausgehen, dass die vorhandenen Schnittstellen kompatibel bleiben.

Software anpassen

Aus den Statistiken der laufenden Produktion, die wir im letzten Teil der Serie eingeführt haben, gibt es erste Ergebnisse, anhand derer das Team bereits Trends ableiten kann. Sie können ausgewertet werden, damit stetig Verbesserungen oder neue Features eingebracht werden können, die auf verschiedene Art und Weise zu klassifizieren sind:

- Bugfix an einem Feature aufgrund von dringenden Anpassungen, die ansonsten entweder die Stabilität gefährden oder fachlich nicht korrekt arbeiten.
- Changes an einem bereits laufenden Feature aufgrund von Anpassungen, die nicht dringend sind.
- Erweiterung des Featureumfangs durch neue Services.

Wichtig an dieser Stelle ist auch, dass sich die Entwickler bereits in einem laufenden Entwicklungsprozess befinden. Diesen müssen sie bei Changes oder Bugfixes verlassen, insbesondere, wenn sie als Hotfix sofort ins System müssen. Der Entwickler muss also seine aktuelle Arbeit unmittelbar unterbrechen. In den seltensten Fällen wird seine aktuelle Entwicklung dabei auf einem sauberen Stand sein, den er ohne Probleme committen kann. Um diesen Teil, der gerade in Arbeit ist, trotzdem zu sichern, um später daran weiterzuarbeiten, bietet es sich an, ihn mittels *git stash* zu parken. Danach kann dann in Ruhe der Bug gefixt, und nach Abschluss können die zuvor geparkten Änderungen mittels *git stash pop* wieder zurückgeholt werden.

Um den Bugfix oder auch komplette Features von einem Branch in einen anderen zu kopieren – wenn es sich nur um einen oder wenige Commits handelt – nutzt man *git cherry-pick <commit>*, um einen fremden *commit* in den aktuellen Branch mit zu übernehmen. Dadurch kann der Bugfix aus dem Release-Branch auch in den aktuellen Entwicklungs-Branch übernommen werden oder aber auch umgekehrt ein schon fertiges Feature aus dem Entwicklungs-Branch für das nächste Release.

Fixen eines Bugs

Durch die Analyse unserer gesammelten Daten haben wir zum Zeitpunkt t_1 festgestellt, dass unsere Anwendung beim Aufruf der Fuhrparkübersicht mit der Zeit immer

Abb. 1: Anwendung zur SYNGO-Autovermietung

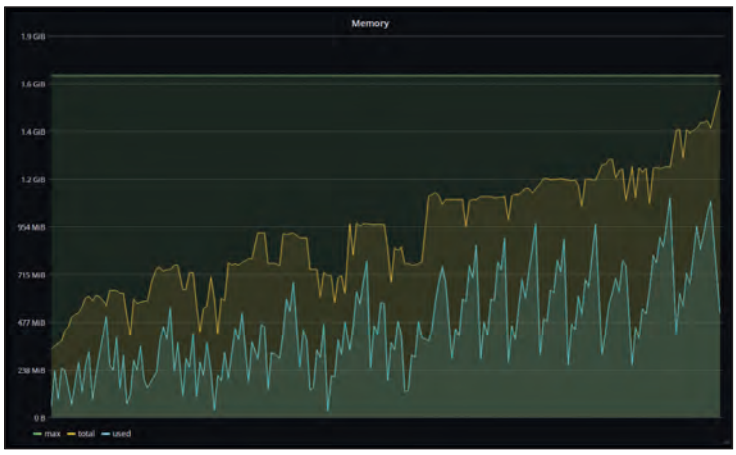

Abb. 2: Speicherstatistik

mehr Speicher verbraucht. Auf dieser Seite werden alle unsere Fahrzeuge angezeigt, wobei deren Zahl konstant ist. Weiterhin zeigt die Seite eine kleine Statistik, wie oft ein konkretes Fahrzeug vermietet wurde (**Abb. 2**).

Das Speicherproblem wurde genauer analysiert und man hat festgestellt, dass die Berechnung der Statistik das Problem für das Speicherloch ist. Naiv werden hier mittels des O/R Mappers alle alten Buchungen von der Datenbank geladen, nur um danach die ermittelte Liste nach ihrer Größe zu fragen. Da der Speicherverbrauch mit steigender Datenmenge weiter ansteigen wird, muss hierfür ein Bugfix bereitgestellt werden. Folgende Schritte sind nun zu tun (**Abb. 3**):

- Auf Basis des Releasetags in Git einen Branch erstellen (falls nicht bereits getan oder vorhanden -> 1.0.x Bugfix). Innerhalb dieses Branches wird dann der Bugfix erstellt.
- Idealerweise wird zunächst ein Test erstellt, der das fehlerhafte Verhalten aufzeigt („Test ist rot“).
- Fixen des Bugs („Test ist grün“).
- Commit aller Änderungen in Git (spätestens hier, gegebenenfalls auch schon zuvor).
- Durchführen der automatisierten Tests (Unit, Integration, UI).

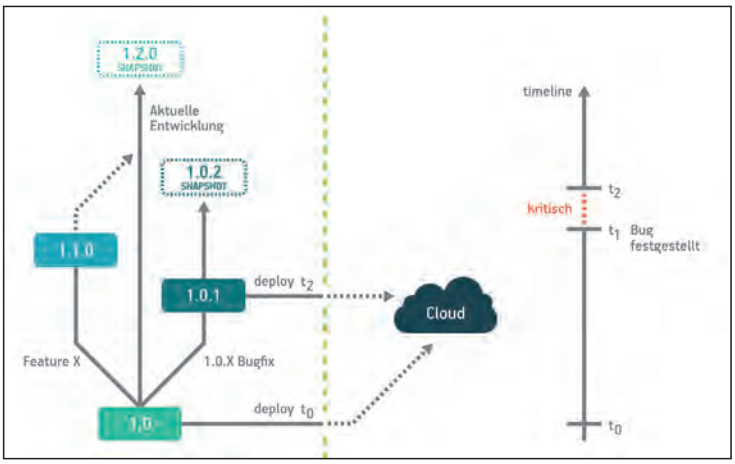

Abb. 3: Zusammenspiel der verschiedenen Branches

- Erstellen eines Releases (inkrementieren der Bugfix-Version [1.0.1], taggen in Git).
- Einspielen in Produktion (Zeitpunkt t_2).

Nicht vergessen: alle Änderungen, die auf dem Bugfix Branch erfolgt sind, müssen auch in die aktuelle Entwicklung – und gegebenenfalls weitere Entwicklungszweige – mit übernommen werden. Sonst taucht der Bug beim nächsten Major-Release gleich wieder auf. In Git kann das mittels Cherry-Picking erfolgen. Eventuell müssen in der aktuellen Entwicklung noch einige Anpassungen vorgenommen werden (abhängig davon, wie weit sie von der vorherigen Version abweicht).

Change

Innerhalb der Fahrzeugübersicht existiert in Version 1.0 der Fuhrparkverwaltung eine einfache *frei/belegt*-Anzeige. Diese gibt grundsätzlich an, ob ein Fahrzeug zum aktuellen Zeitpunkt verfügbar ist oder eben nicht. Der Kunde meldet, dass diese Anzeige leider nur teilweise hilfreich ist. In unseren gesammelten Daten können wir das auch klar erkennen: Die Buchungsseite für ein Fahrzeug wird viel häufiger aufgerufen, als eine konkrete Buchung durchgeführt wird, da das Fahrzeug jetzt verfügbar ist, aber nicht für den gesamten gewünschten Zeitraum. Daher soll das Feature der Verfügbarkeit erweitert werden, sodass mit angezeigt wird, für welchen Zeitraum ein Fahrzeug frei bzw. belegt ist. Letztendlich ist die Vorgehensweise bei der Umsetzung eines Changes die gleiche wie bei einem Bugfix. Der wesentliche Unterschied ist dabei lediglich der Umfang. Es wird sicherlich mehr geändert werden müssen, daher wird es auch mehr Commits geben. Es kann also genauso auf einem separaten Branch („Feature X“) gearbeitet werden, von dem dann das Release erstellt wird. Der Unterschied ist hier nur, dass bei dem Release nicht die Bugfix- sondern die Minor-Version angepasst wird. In unserem Beispiel haben wir dann also die Version 1.1.0. Allerdings kann es bei einem Change auch vorkommen, dass die vom Kunden gewünschte Änderung in der aktuellen Entwicklung bereits entwickelt wurde, jedoch erst für das nächste Release geplant war. Dann kann dieser Change natürlich in umgekehrter Richtung aus der aktuellen Entwicklung in den Change-Branch übernommen werden.

> **Versionen**
>
> Zur Bildung von Versionsnummern gibt es keine festen Regeln. Wichtig ist letztlich nur, dass man seine eigenen Konventionen hat, sie einhält und sich darauf verlassen kann. Das erleichtert insbesondere das Arbeiten mit mehreren Teams, wenn genau geregelt ist, welche Versionsänderung welche Bedeutung hat, und welche Auswirkungen das auf die weitere Entwicklung hat. Auf jeden Fall sollte man sich daran halten, dass ein Artefakt mit einer festen Versionsnummer genau in einer Ausführung existiert, d. h. wenn einmal ein Release erstellt wurde, wird daran nichts mehr geändert. Man kann daher davon ausgehen, dass, egal wo man das Artefakt findet, es immer den gleichen Inhalt hat.

Entwicklung eines neuen Features

Als neues Feature für die kommende Version sollen innerhalb der Fahrzeugübersicht für jedes Fahrzeug bestimmte Eigenschaften wie die Anzahl der Sitzplätze oder die Kofferraumgröße durch Icons dargestellt werden. Unsere Statistik zeigt dabei, dass die Fahrzeugübersicht der zentrale Einstiegspunkt innerhalb der Anwendung ist, mit der am meisten gearbeitet wird. Daher muss hier besonders darauf geachtet werden, dass diese Seite weiterhin sehr schnell geladen werden kann.

Die Entwicklung eines neuen Features geschieht innerhalb der normalen Weiterentwicklung. Daher kann es gut sein, bei einem Release nicht nur ein Feature, sondern gleich mehrere Features mit zu releasen. Einzelne Features können innerhalb des Releases mit Feature-Flags ausgestattet werden, sodass sie auch im Livebetrieb aktiviert bzw. deaktiviert werden können. In unserem Beispiel wäre das die Anzeige der Eigenschaft „sichtbar“ oder „nicht sichtbar“. Über die Feature-Flags ist es auch möglich, dass ein Feature nur einer bestimmten Nutzergruppe zur Verfügung gestellt wird, um es mit dieser Nutzergruppe zunächst im Livebetrieb zu erproben. Allerdings sollte man bei der Verwendung solcher Feature-Flags immer bedenken, dass man sie auch wieder aus dem bestehenden Code ausbauen sollte, falls das Feature erfolgreich ausgerollt wurde. Passiert das nicht, wird auf Dauer einerseits der Code und andererseits die Anzahl der Feature-Flags sehr unübersichtlich. Wahrscheinlich ist nach einiger Zeit auch nicht mehr hundertprozentig klar, welche Abhängigkeiten zwischen den Features bestehen. Der oben genannte Bug des ansteigenden Speicherverbrauchs ist ein gutes Beispiel dafür, was durch das aktive Monitoring der Produktionssoftware erreicht werden kann. Fände kein Monitoring statt oder wertete man es nicht aus, würde dieses Speicherloch irgendwann dazu führen, dass die Anwendung überlastet wird. Durch das Monitoring

kann der Bug jedoch schon erkannt werden, bevor es eine tatsächliche spürbare Auswirkung für den Endanwender gibt. Dadurch wird unsere Software insgesamt stabiler, da wir Fehler bereits beheben können, bevor sie zu einem richtigen Problem werden.

Chaos Monkey

Ist die Software in der Produktion und die Entwickler sind nicht voll mit der Weiterentwicklung beschäftigt, ist das der perfekte Zeitpunkt, sich weiter um die Stabilität zu kümmern. Das bedeutet, dass man die gesammelten Daten auswertet, um z. B. Bugs zu erkennen, wie den hier genannten. Weiterhin kann man aus den Daten Trends ablesen und so frühzeitig die Weichen für die weitere Entwicklung stellen beziehungsweise den Betrieb darauf ausrichten. Und letztlich kann man in diesen Zeiten die Anwendung auch richtig unter Stress setzen, konkret: einzelne Teile der Anwendung (zufällig) herunterfahren. Im Idealfall sollte unsere Anwendung so stabil sein, dass das keine Auswirkungen auf das Gesamtsystem hat; der Endanwender sollte davon also nichts mitbekommen. Auch kann für diese Fälle das Monitoring analysiert, erweitert und verbessert werden. Durch das konsequente Monitoring unserer Software haben wir die Möglichkeit geschaffen, Trends zu erkennen und einzugreifen, bevor es zu spät ist. Weiterhin liefern uns diese Daten wertvolle Informationen, wie und wo wir unsere Software weiterentwickeln müssen. Dem Kunden kann dadurch eine deutlich stabilere Anwendung zur Verfügung gestellt werden, was letztlich zu einer höheren Zufriedenheit auf beiden Seiten führt.

Fazit

Beim Versuch, ein abschließendes Fazit über die Artikelserie zum Thema Produktion zu ziehen, ist uns aufgefallen, dass es auch nun wieder kein Ende gibt. Wir haben immer noch Lücken in den Themen Konfigurationsmanagement und im Bereich der aktuell immer stärker aufkommenden Containerlandschaft durch Werkzeuge wie Docker. Auch neue Trends wie Serverless können hier komplett neue Möglichkeiten bereitstellen. Den eigentlichen Lebenszyklus eines Softwaresystems haben wir aber abdecken können.

Joachim Arrasz ist als Software- und Systemarchitekt in Karlsruhe bei der synyx GmbH & Co. KG als Leiter der CodeClinic tätig. Darüber hinaus twittert und bloggt er gerne.

@arrasz **http://blog.synyx.de/**

Sebastian Heib ist als Softwareentwickler in Karlsruhe bei der synyx GmbH & Co. KG tätig und beschäftigt sich dort mit der Entwicklung verteilter Backend-Systeme und CodeClinic-Aufgaben.